# CloudNativePG: Robust, Self-Healing PostgreSQL on Kubernetes

Jimmy Angelakos
Staff Software Engineer, pgEdge

SCaLE 23x
2026-03-06

pgEdge®

# About Jimmy Angelakos

- Systems & Database Architect, based in Edinburgh, Scotland
- Involved in the Open Source Community for 25+ years
- PostgreSQL Significant Contributor
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- Co-author, PostgreSQL 16 Administration Cookbook
- pg_statviz PostgreSQL extension

# What is this talk about?

- CloudNativePG (CNPG)

  (duh)

  - History
  - Design philosophy
  - Features and capabilities
- The lifecycle of databases vs cloud-native databases
- The ease of deployment

# What is this talk not about?

- Running things vs running things on Kubernetes
- In-depth examination of CloudNativePG use cases
- CloudNativePG deep dive
  - Or coding

# What is CloudNativePG?

- Open source Kubernetes *operator* for PostgreSQL
  - Like a human operator running a software service
  - Uses *custom resources* (extensions of K8s API)
- Accepted into the Cloud Native Computing Foundation (CNCF) Sandbox
- Manages the full lifecycle of *highly available* PG clusters
- Declarative and K8s-native

# A bit of history 🏛️

- CNPG originated at a PostgreSQL company called 2ndQuadrant (now part of EDB)
- My (small) involvement: part of initial planning discussions
- Project has since grown far beyond its origins
  - True community effort
  - Earned a place in CNCF Sandbox
  - Thriving ecosystem of contributors and companies (incl. pgEdge)

CloudNativePG

# Why run Postgres on K8s?

- The rest of your application stack is on K8s (duh)
- Infrastructure as Code (define in YAML, version in Git)
- Deployment automation ensures consistency (dev/staging/prod)
- You get:
  - Automated failover, self-healing, scaling
  - K8s scheduling, networking, observability

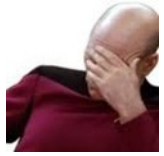
# How to run Postgres on K8s?

- "But PostgreSQL is a 30 year old database"
- CNPG embraces K8s, doesn't fight it:
  - Declarative vs imperative: describe the *end state we want*, not *how*
  - Postgres concepts are mapped directly to K8s Custom Resource Definitions (CRDs)
- You don't manage individual pods: just deploy a *Cluster* manifest
  - `kubectl apply -f mycluster.yaml`
- Don't treat DB instances like delicate flowers
  - If one breaks, you grow a new one from the seed (the YAML)

# Postgres concepts → K8s CRDs

| PostgreSQL concept | CloudNativePG resource |
|---|---|
| Database cluster | Cluster |
| Backup | Backup |
| Scheduled (cron) backup | ScheduledBackup |
| Connection pooler | Pooler |
| Logical Replication Publication | Publication |
| Logical Replication Subscription | Subscription |

# Cluster…



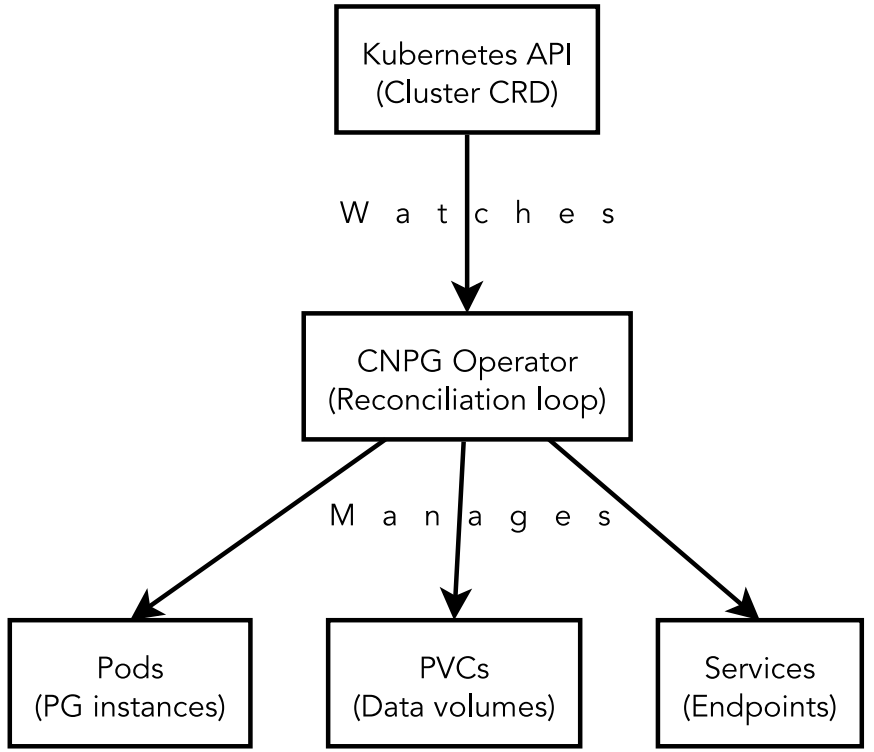
- In the PostgreSQL Documentation, a *cluster* is a single data directory
  - A PostgreSQL server instance
- In the IT/database industry, a *cluster* is a group of servers
  - Working together for scaling or HA purposes
- In CloudNativePG, a *Cluster* CRD is a single unit of management
  - Represents a primary instance and its standby replicas
  - Opinionated: just one database per Cluster
    - Simplifies lifecycle management, backups, resource isolation
    - Better alignment with microservices philosophy

# CNPG design philosophy (i)

## Design

- Does not use `StatefulSets` like other operators
- Manages Pods and `PersistentVolumeClaims` directly
  - Granular control over instance & storage lifecycle

## Operator pattern



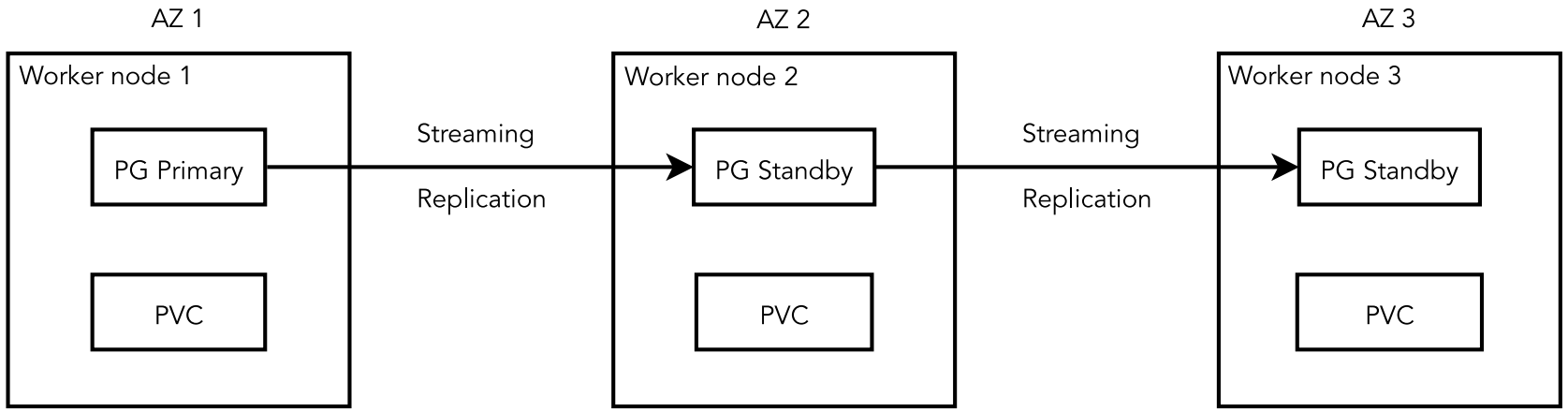
# CNPG design philosophy (ii)

- Each PG pod runs the *Postgres instance manager* as PID 1
  - Manages PG server process (`postmaster`): no Patroni, RepMgr etc.
  - Probes startup, liveness, readiness on behalf of the operator
- Uses PG native physical streaming replication based on the Write-Ahead Log (WAL) – not k8s storage-level replication
  - Extremely robust and battle-proven
  - File-based log shipping as fallback (e.g. WAL files to object store)
  - "Hot Standby": Replicas actively serve read-only workloads

# Shared-Nothing architecture

## Core PostgreSQL resilience tenet

- In Kubernetes:
  - PG instances on different worker nodes, with dedicated storage
  - Only network is shared, spread across 3+ availability zones



# Services: the “killer feature”

- `-rw` services route to current primary only (RW workload)
- `-ro` services route to hot standbys only (RO offload)
- `-r` services route to any instance (don’t use this)
- Automatic reconfiguration upon failover
- `ClusterIP` by default, but can be `LoadBalancer`, `NodePort`, etc.

# Enough talk. How do I do it?


I GOTTA HAVE
MORE YAML!
memegenerator.net

# My first Cluster (i)

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: my-first-cluster
spec:
  instances: 3
  storage:
    size: 10Gi
  postgresql:
    parameters:
      shared_buffers: "256MB"
      max_connections: "100"
```

# My first Cluster (ii)

```
kubectl apply -f cluster.yaml
kubectl get cluster
```

- Each instance gets its own PVC
- Storage is entirely dependent on your `StorageClass` choice

```
spec:
  storage:
    size: 50Gi
    storageClass: my-storage-class
```

# Backups (i)

- CNPG-I Plugin Interface: extensible system for storage and backups
  - Primary implementation: Barman Cloud Plugin (works with object stores)
  - Continuous WAL archiving for Point-In-Time Recovery (PITR)

```
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: daily-backup
spec:
  schedule: "0 0 2 * * *"  # 2 AM daily (note format)
  cluster:
    name: my-first-cluster
  method: plugin
  pluginConfiguration:
    name: barman-cloud.cloudnative-pg.io
```

# Backups: restoring (i)

- **Never** restore a backup to the same cluster
  - Restoring in-place risks corruption and conflicts
  - Make a fresh `Cluster`, bootstrapped from the backup
  - Remember: your clusters are *reproducible*

# Backups: restoring (ii)

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: my-restored-cluster
spec:
  instances: 3
  storage:
    size: 10Gi
  bootstrap:
    recovery:
      source: backup-source
```

# Yes but do you even PROD?

# Migration (i)

- `externalClusters` is just a conf block (a connection definition)
- Import modes:
  - `microservice`: each database in a separate `Cluster`
    - Owned by the `initdb.owner` user
  - `monolith`: entire (PG) *cluster* (of Dbs) in one (CNPG) `Cluster`
    - Anti-pattern
- Your source PG DB needs zero special configuration
- Migrate from VMs, on-prem, RDS, Aurora, etc. via `pg_dump`/`pg_restore`
  - Anything with a PG connection that you can `pg_dump` from

# Migration (ii)

```
spec:
  instances: 3
  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - "frogge"
        source:
          externalCluster: source-db
  externalClusters:
    - name: source-db
      connectionParameters:
        host: old-pg.frogge-emporium.com
        user: postgres
        dbname: frogge
      password:
        name: source-db-credential
        key: password
```

- With logical replication, you can live migrate + major version upgrade

# Connection pooling (i)

- `Pooler` CRD
- PgBouncer as a separate deployment (not sidecar)
- Routes through CNPG k8s services
- Scales independently of DB deployment
- Transparent TLS
- Prometheus metrics built-in

# Connection pooling (ii)

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-rw
spec:
  cluster:
    name: my-cluster
  instances: 3
  type: rw
  pgbouncer:
    poolMode: transaction
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

# Replica clusters

- For HA/DR and R-O offloading use
- “Designated primary” in DR k8s cluster receives WAL updates, ready to take over
- Multi-region, hybrid, multi-cloud
- Cross-cluster failover manually coordinated
  - Demote first, promote next
  - No split-brain by design

# Rolling upgrades

- Minor version upgrades (i.e. 17.1 → 17.2)
  - Automatically handled by CNPG
  - Standbys first → Switchover → Old primary
- Major version upgrades (i.e. 16.5 → 17.1)
  - Handled by CNPG, you update `imageName` in `Cluster`
  - Runs `pg_upgrade` with hard links
  - Standbys re-cloned from upgraded primary
- Uninterrupted, with Spock* bi-directional logical replication

# pgEdge + CNPG

- Multi-master with Spock
- CNPG handles HA *within* each k8s cluster
- Spock handles active-active logical replication *between* k8s clusters
  - 2+ CNPG `Clusters`, each with a primary & standbys
  - Both accept writes simultaneously
  - Conflict resolution configurable (last-update-wins, etc.)
  - Conflict avoidance (delta-apply columns for counters/balances)

# pgEdge Helm chart

- Automates the full setup: CNPG clusters + Spock replication + init jobs
- `helm install` → active-active out of the box
- Handles replication slot recreation on major version upgrades
- Open source: github.com/pgEdge/pgedge-helm

```
helm repo add pgedge https://pgedge.github.io/pgedge-helm
helm install pgedge pgedge/pgedge \
  --values examples/configs/single/values.yaml \
  --wait
```

# Let’s do it

# Quick start from scratch (i)

- Prerequisites (kind: k8s in docker)

```
sudo apt -y install kind kubectl
sudo dpkg -i kubectl-cnpg_1.28.1_linux_x86_64.deb
```

- Create kind cluster

```
kind create cluster --name cnpg-demo
kubectl cluster-info --context kind-cnpg-demo
```

- Install CNPG

```
kubectl apply --server-side -f cnpg-1.28.1.yaml
```

- Verify installation

```
kubectl rollout status deployment \
    -n cnpg-system cnpg-controller-manager
```

# Quick start from scratch (ii)

- Define the cluster

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: my-first-cluster
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:18
  storage:
    size: 500Mi
```

- Deploy the cluster

```
kubectl apply -f my-first-cluster.yaml
```

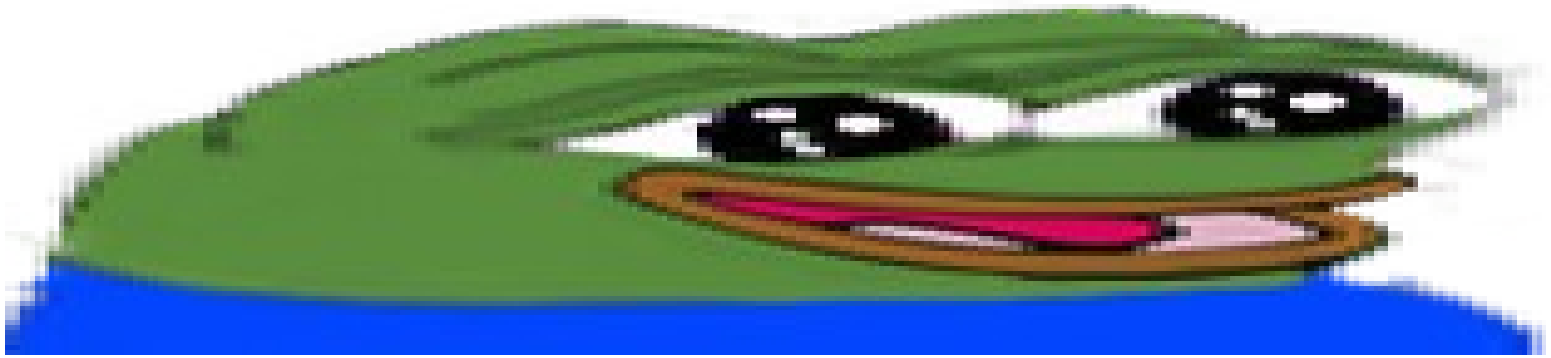
- Verify deployment

```
kubectl get pods -l cnpg.io/cluster=my-first-cluster
```

# Quick start from scratch (iii)

- Check in more detail using the plugin

```
kubectl cnpg status my-first-cluster
```



# Quick start from scratch (iv)

- Find the service to connect to

```
kubectl get svc | grep my-first-cluster
```

- Find the endpoint to connect to

```
kubectl get endpoints my-first-cluster-rw
```

# Let’s break it

# Let's observe what happens...

- Top pane: observability

```
watch -n0.5 "kubectl get pods \
    -l cnpg.io/cluster=my-first-cluster -o wide \
    && echo \
    && kubectl cnpg status my-first-cluster"
```

- Bottom pane: chaos
  - You will see things reconfiguring pretty quickly

```
kubectl delete pod my-first-cluster-1 --wait=false
```

# Verify recovery

- Bottom pane:

```
kubectl get endpoints my-first-cluster-rw
```

# Some final notes

# Image volume extensions

- Mount extension images directly
  - No custom builds needed
- Extensions mounted as volume points at pod startup
- Base PG image stays clean (and updatable)

# Argo CD for multi-k8s deployments

- GitOps: entire topology lives in Git
- Argo CD deploys the Helm chart to each cluster
- Switchover/failover are just Git commits

# Thank you!

Discount code **scale23x** for **45% off** all products